ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 44, No. 4, pp. 363-373, 2017. 4
https://doi.org/10.5626/JOK.2017.44.4.363

# 커널 버전 별 Ext4 파일 시스템의 fsync()에 대한 고찰

## (Consideration of fsync() of the Ext4 File System According to Kernel Version)

손 성 배 [†] 노 연 진 [††] 이 도 근 [†††] 박 성 순 [††††] 원 유 집 [†††††]
(Seongbae Son) (Yoenjin Noh) (Dokeun Lee) (Sungsoon Park) (Youjip Won)

**요 약** Ext4 파일시스템은 리눅스를 기반으로 하는 PC, 서버, 임베디드 시스템 등에 널리 사용되고 있고, 동작 분석 및 성능 향상에 대한 많은 연구들이 진행되어왔다. 하지만, 버퍼를 활용한 입출력을 사용하는 Ext4에서 특정 파일에 대한 즉시 내구성을 요구하는 fsync() 시스템 콜을 커널 버전 별로 자세히 분석한 연구는 찾아보기 힘들다. fsync() 시스템 콜의 분석을 통해 커널 3.4.0 ~ 4.6.4 버전 중 3.4.0, 3.8.0, 그리고 4.6.2 세 개의 버전에서 fsync() 동작에 차이가 있음을 발견하였다. 커널 3.4.0 버전은 3.7.10 이후 버전에 비해 긴 지연시간을 가지는 특징이 있다. 3.8.0 버전은 Ext4 저널링의 순차 모드가 깨질 수 있는 단점을 가지고 있으며, 해당 문제는 4.6.2 버전에서 해결되었다.

**키워드:** fsync() 시스템 콜, Ext4 파일 시스템, 블록 입출력, 블록 트레이스, 저널링

***Abstract*** Ext4 file system is widely used in various computing environments such as those of the PC, the server, and the Linux-based embedded system. Ext4, which uses a buffer for block I/O, provides fsync() system call to applications to guarantee the consistency of a specific file. A log of the analytical studies regarding the operation of Ext4 and the improvement of its performance has been compiled, but it has not been studied in detail in terms of kernel versions. We figure out that the behavior of fsync() system call is different depending on the kernel version. Between the kernel versions of 3.4.0 and 4.7.2, 3.4.0, 3.8.0, and 4.6.2 showed behavioral differences regarding the fsync() system call. The latency of fsync() in kernel 3.4.0 is longer than that of the more-advanced 3.7.10; meanwhile, the characteristics of 3.8.0 enabled the disruption of the Ext4 journaling order, but the ordered defect was solved with 4.6.2.

**Keywords:** fsync() system call, Ext4 file system, block i/o, block trace, journaling

· 본 연구는 산업통상자원부 및 한국산업기술평가관리원의 산업원천기술개발사업(정보통신)의 일환으로 수행하였습니다. [No.10041608, 차세대 메모리 기반의 스마트 디바이스용 임베디드 시스템 소프트웨어]
· 본 연구는 미래창조과학부 및 정보통신기술진흥센터의 정보통신·방송 연구개발사업의 일환으로 수행하였습니다. [R7117-16-0232, 32Gbps 데이터 서비스를 위한 익스트림 스토리지 입출력 기술 개발]

† 학생회원 : 한양대학교 컴퓨터.소프트웨어학과
afireguy@hanyang.ac.kr
†† 정 회 원 : 한양대학교 컴퓨터.소프트웨어학과
maguyn@hanyang.ac.kr
††† 비 회 원 : 한양대학교 컴퓨터.소프트웨어학과
matureelf@hanyang.ac.kr
†††† 종신회원 : 안양대학교 컴퓨터공학과 교수
sspark@anyang.ac.kr
††††† 종신회원 : 한양대학교 컴퓨터.소프트웨어학과(Hanyang Univ.)
yjwon@hanyang.ac.kr
(Corresponding author임)

논문접수 : 2016년 11월 11일
(Received 11 November 2016)
논문수정 : 2016년 12월 12일
(Revised 12 December 2016)
심사완료 : 2016년 12월 27일
(Accepted 27 December 2016)

Copyright©2017 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.
정보과학회논문지 제44권 제4호(2017. 4)

## 1. 서 론

Ext4[1]는 현재 리눅스 커널의 기본 파일시스템으로, 안드로이드 스마트폰과 같은 모바일 기기나 데스크톱 컴퓨터에 널리 사용되고 있다[2]. Ext4의 fsync()의 지연시간은 애플리케이션 성능과 밀접한 관련이 있기 때문에, fsync()에 대한 구조 분석[3] 및 성능 개선[4-9]에 대한 활발한 연구가 진행되어 왔다.

Jeong et al. [5]은 Asynchronous I/O로 발행되었지만 synchronous I/O의 property를 가지는 QASIO(Quasi-Asynchronous I/O)라 정의된 Asynchronous I/O를 다른 Asynchronous I/O 보다 우선적으로 처리되도록 하여 fsync() 및 파일 시스템의 응답성을 개선하는 기법을 제안했다. Kang et al. [6]은 DRAM과 NVRAM을 동시에 적용한 하이브리드 메모리 아키텍처에서 더티 페이지를 NVRAM에 복사하는 FS-LRU 페이지 Cache 알고리즘을 통해 fsync()의 동기화 비용을 줄이는 기법을 제안하였다. Kang et al. [7]은 블록 그룹에 따라 저널링(per-block-group Journaling)을 수행하여 fsync()의 지연시간을 줄이는 기법을 제안한다. Park et al. [8]은 Command Queuing을 지원하는 UFS(universal flash storage)나 SSD(solid state disk)와 같은 낸드 플래시 메모리를 사용하는 시스템에서 유저 애플리케이션이 fsync()를 호출하는 시점을 예측하고, Queue Depth를 줄이는 Queue Depth Adaptation 기법 적용하여 fsync()의 지연시간을 줄이는 기법을 제안하였다. 이러한 연구들은 fsync()의 성능을 개선하였지만, 리눅스 커널에서 fsync() 동작에 대한 자세한 분석은 포함하고 있지 않다.

본 논문에서는 리눅스 커널 버전 3.4.0에서 4.6.4 사이에서 fsync() 동작의 차이가 있는 3.4.0, 3.8.0, 4.6.2 세 개 버전의 fsync()에 대해 이들을 심도있게 분석하고 동작 메커니즘의 특징을 파악한다. 해당 분석을 통해서, 메커니즘의 차이가 성능에 미치는 영향을 분석한다. 3.4.0 버전의 fsync()는 3.7.10 이후 버전에 비해 처리시간이 길다. 3.8.0 버전은 Ext4 저널링의 Ordered Mode를 보장하지 못하는 경우가 발생한다. 저널링 구현상의 버그이다. 해당 버그는 4.6.2 버전의 fsync()에서 해결되었다.

## 2. 배 경

### 2.1 블록 입출력

블록 디바이스 계층(generic block device layer)에서는 2.5.1 버전부터 도입된 'bio'라는 자료구조가 핵심이다. 블록 입출력은 bio 구조체를 구성하는 것으로 시작된다. bio 구조체는 블록 디바이스 계층에서 하나의 입출력 연산에 대한 정보를 가지는 자료구조이다. bio 구조체의 구성은 그림 1과 같다. include/linux/blk_types.h에 정의되어 있다.

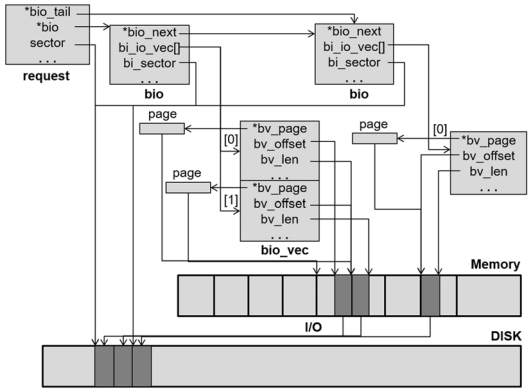


그림 1 Request와 bio 구조
Fig. 1 Layout of request and bio

파일 데이터는 물리 메모리에 불연속하게 저장 될 수 있다. 블록 연산을 위해서는 이러한 불연속하게 저장된 물리 메모리 공간을 하나의 연속된 단위로 관리할 필요가 있다. 불연속하게 저장된 각각의 메모리 공간을 '세그먼트'[10]라 하고, 각 세그먼트는 bio_vec 자료 구조를 통해 관리된다. bio 구조체는 bio_io_vec 필드를 통해 bio_vec 자료 구조를 배열로 저장하여 입출력 연산과 관련된 세그먼트들을 관리한다.

bio 구조체의 bi_sector 필드는 디스크 상에서 입출력 연산의 대상이 되는 시작 주소를 가리킨다. bi_sector 필드는 bio가 초기 생성될 때 논리 블록 주소(LBA)로 설정되지만, request에 병합되기 전에 물리 블록 주소(PBA)로 변경된다. 논리 블록 주소는 디스크 파티션 내의 블록 인덱스를 의미하며, 물리 블록 주소는 실제 디스크 내에서의 블록 인덱스를 의미한다. LBA 0는 해당 파티션에서 첫 번째 블록이다. 물리적으로는 디스크의 첫 블록이 아닐 수 있다.

bio_vec 구조체는 bv_page 필드를 통해 세그먼트가 포함된 페이지를 가리키며, bv_offset과 bv_len 필드를 통해 페이지 내의 세그먼트의 시작 위치와 크기를 나타낸다.

이렇게 구성된 bio는 submit_bio() 함수를 통해 일반 블록 계층인 plugged list 또는 입출력 스케줄러에 전달된다. 해당 bio가 입출력 스케줄러에서 관리하는 엘리베이터 큐(Elevator Queue)에서 병합 가능하면 병합하고, 그렇지 않으면 프로세스마다 존재하는 plugged list에 삽입한다. Plugged list와 엘리베이터 큐는 request들로 구성된다.

리눅스의 블록 디바이스 계층은 인접한 블록을 위한 요청을 모을 수 있는 기회를 높이기 위해 '플러깅(Plugging)'과 '언플러깅(Unplugging)'이라고 알려진 기술을 사용한다. '플러깅'이란 생성된 요청을 plugged list에 삽입하는 것을 의미한다. 모든 프로세스는 자신의 plugged list를 가지며, 요청을 입출력 스케줄러에 전달하기 전에 자신의 plugged list에 우선 삽입함으로써 일정 시간동안 인접한 블록의 요청들이 병합되도록 한다. 이렇게 되면 여러 개의 요청이 병합되어 하나의 요청으로 처리되기 때문에 전체 입출력의 성능을 향상시킬 수 있다. '언플러깅'은 plugged list에 있는 요청을 입출력 스케줄러에 전달하는 것을 의미한다.

엘리베이터는 입출력 스케줄러의 추상화 계층이며, 디스크 헤드가 트랙에서 한쪽 방향으로 움직이도록 요청을 엘리베이터 큐에 정렬하기 때문에 엘리베이터라고 부른다.

Request는 인접한 블록을 위한 연산들을 모을 수 있는 기회를 높임으로써, 전체 블록 입출력 성능이 향상되도록 한다. Request는 그림 1에 보이는 것과 같이 한 개 이상의 bio를 가지며, 리스트로 관리한다. Request 구조체의 bio 필드와 biotail 필드는 각각 첫번째와 마지막 bio를 가리킨다. Request 구조체의 sector 필드는 Request에 포함된 첫 번째 bio의 bi_sector 값을 가지며, 디스크 상에서 해당 Request의 입출력 대상이 되는 시작 주소이다. 장치 드라이버(Device Driver)는 Request의 sector 필드를 통해 실제 장치와 입출력 작업을 수행한다.

그림 2는 블록 계층에서 Request의 전달 경로를 나타낸다. Plugged 리스트에 포함된 Request들은 언플러그 시키는 함수가 호출되거나 프로세스 스케줄러에 의해 문맥 교환(Context Switch) 시 언플러그 되고, 입출력 스케줄러에 전달되어 엘리베이터 큐에 삽입된다. 여기서 언플러그는 plugged list에 있는 요청을 입출력 스케줄러에 전달하는 것을 의미한다.

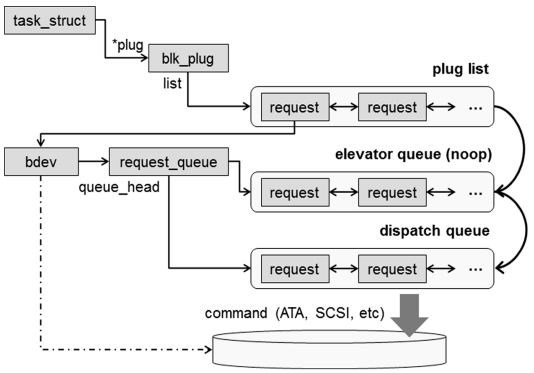


그림 2 블록 I/O 흐름
Fig. 2 Block I/O flow

입출력 스케줄러에 전달된 요청들은 입출력 스케줄러의 정책에 의해서 디바이스의 디스패치 큐(Dispatch Queue)[11]에 삽입된다. 디스패치 큐 내의 요청들은 FIFO 방식으로 디바이스 드라이버에 의해 처리된다.

### 2.2 I/O 연산의 종류

입출력 연산은 submit_bio() 함수를 호출하는 프로세스가 최종적으로 입출력 스케줄러에 요청을 전달하는 것이다. submit_bio() 함수는 시스템 콜이나 flusher 쓰레드, swapper 쓰레드와 같은 커널 쓰레드에 의해 수행된다.

입출력 연산의 종류는 READ, WRITE, READA, READ_SYNC, WRITE_SYNC, WRITE_FLUSH, WRITE_FUA, WRITE_FLUSH_FUA 등이 있다. 입출력 연산의 종류는 include/linux/fs.h에 정의되어 있다.

READ와 WRITE는 읽기와 쓰기를 의미하며, SYNC는 입출력 스케줄러(I/O scheduler)에 따라 다르게 동작될 수 있다. 예를 들어, CFQ 입출력 스케줄러는 Synchronous 큐와 Asynchronous 큐를 통해 Synchronous와 Asynchronous 요청을 각각 관리하며, Synchronous 큐의 우선순위가 더 높기 때문에 SYNC 플래그가 설정된 Synchronous 요청이 우선적으로 처리된다. 하지만, Synchronous와 Asynchronous 큐를 따로 관리하지 않는 NOOP와 같은 입출력 스케줄러는 SYNC 플래그를 설정하는 것이 의미가 없다.

FLUSH는 디스크에 write-back 캐시가 있는 경우 캐시에 있는 요청을 디스크 표면에 기록하는 것을 의미하고, FUA(Force Unit Access)는 write-back 캐시를 거치지 않고 바로 디스크에 기록하는 것을 의미한다. 따라서 FLUSH가 설정된 요청을 통해 FLUSH 요청이 발생하기 전후 요청들의 순서를 보장해줄 수 있다.

### 2.3 Scatter-gather DMA

DMA(Direct Memory Access)는 입출력 장치와 호스트 컴퓨터 간에 데이터를 송수신 하는 방법의 일종이다. 호스트 컴퓨터가 저장장치와 데이터를 송신(혹은 수신)하는 방법에는 두가지가 있다. Programmed IO와 DMA이다[12]. scatter-gather[13] DMA는 하나의 쓰기 명령어를 통해서 메모리상에서서 떨어져 있는 여러 세그먼트들을 저장장치에 기록하는 기법이다.

그림 1과 같이 각 디바이스에 전달되는 Request 요청은 하나 이상의 bio를 인자로 갖을수 있다. bio는 하나이상의 세그먼트를 인자로 갖는다. 블록 장치 드라이버는 저장장치가 scatter-gather DMA를 지원하느냐에 따라 다른 방식으로 디스패치 큐의 요청을 처리한다. scatter-gather DMA를 지원하지 않는 블록 장치인 경

우, 요청의 각 bio에서 각각의 세그먼트의 전송이 완료될 때까지 DMA 요청을 반복한다. scatter-gather DMA를 지원하는 블록 장치의 드라이버는 한 번의 DMA 전송을 통해 요청에 포함된 bio들의 세그먼트를 처리한다. 호스트에서 저장장치로 데이터가 모두 전송되면 (쓰기 요청시) 저장장치는 호스트 컴퓨터에게 인터럽트 신호를 전송한다. 인터럽트 신호를 수신하면, 요청 내 모든 세그먼트가 전송되었는지 검사한 후, 전송이 완료되었을 경우 디스패치 큐에서 해당 요청을 제거하고 디스패치 큐 내의 다음 요청을 처리한다.

### 2.4 Ext4의 저널링

2.4.1 Ext4 저널링 Mode

Ext4 저널링[10,14]은 Ordered, Data, Writeback 세 개의 모드를 지원한다. Ordered 모드가 기본이다. Data 모드는 변경된 데이터와 메타데이터를 모두 저널 영역에 기록한다. 세 가지 모드 중 안정성이 가장 높지만, 변경된 데이터와 메타데이터가 저널 영역과 데이터 영역에 두 번 기록되기 때문에 가장 느리다. Ordered 모드는 메타데이터만 저널 영역에 기록한다. 메타데이터를 저널 영역에 기록하기 전에 데이터를 데이터 영역에 기록함으로써 파일시스템과 디스크 사이에 일관성을 유지한다. Writeback 모드는 Ordered 모드와 같이 저널 영역에 메타데이터에 대한 로그를 남기지만, 데이터 기록의 순서는 보장하지 않는다. Writeback 모드는 세 가지 모드 중 가장 빠르지만, 안정성은 가장 낮다.

2.4.2 Ext4의 트랜잭션

Ext4 파일 시스템에서 한번에 기록되는 로그정보의 집합을 저널 트랜젝션이라 한다. 하나의 트랜잭션은 디스크립터 블록, 메타데이터 블록, 커밋 블록 세 가지로 구성된다. 트랜잭션에 포함된 메타데이터는 메타데이터 블록으로 구성되고, 메타데이터의 파일 시스템 내에서의 위치 정보는 디스크립터 블록에 저장되어 관리된다. 커밋 블록은 데이터를 포함하지 않으며, 해당 트랜잭션의 커밋 완료를 나타낸다. 현재의 저널링은 효율적인 쓰기를 위해 여러 파일 연산의 결과를 모아서 한꺼번에 기록하는 컴파운드 트랜잭션(Compound 트랜잭션)[8,15]을 사용한다.

Ext4의 트랜잭션 디스크립터는 메타데이터, 아이노드, 그리고 체크포인트 등의 정보를 리스트로 관리한다. 하나의 트랜잭션은 여러 파일 연산에 의해 변경된 메타데이터들이 포함될 수 있는 컴파운드 트랜잭션이기 때문에, 트랜잭션에 포함된 메타데이터들은 리스트로 관리된다. 트랜잭션 디스크립터에 포함된 아이노드 리스트는 순차 모드를 위한 자료구조이다. 순차 모드인 경우 트랜잭션에 포함된 메타데이터를 저널에 기록하기 전에 데이터를 먼저 디스크에 기록해야 한다. 따라서 메타데이터에 해당하는 파일의 아이노드를 트랜잭션의 아이노드 리스트에 추가하고 트랜잭션에 포함된 메타데이터를 저널에 기록하기 전에 아이노드 리스트에 포함된 아이노드를 통해 파일의 데이터를 먼저 디스크에 기록하여 순서를 보장한다.

2.4.3 커밋과 체크 포인트

Ext4 저널링은 파일 시스템에 대한 변경을 커밋(Commit)과 체크 포인트(Check Point) 두 단계를 통해 실행된다. Ext4 저널링에서는 파일 연산이 일어날 때 발생하는 데이터를 실제 스토리지의 위치에 기록하기 전에, 데이터의 복사본을 저널 영역에 먼저 기록한다. 이 기록 과정을 커밋(Commit)이라고 한다. 저널에 대한 입출력 데이터 전송이 완료되면(Commit 되면) 블록을 파일 시스템에 기록하고, 파일 시스템에 대한 입출력 데이터 전송이 완료되면 저널에 있는 블록 복사본을 제거한다. 이 과정을 체크 포인트(Check Point)라고 한다.

2.4.4 JBD2 데몬

Ext4 저널링은 JBD2(Journaing Block Device2) 데몬을 의해 수행되며, 기본적으로 5초마다 동작하거나 fsync() 시스템 콜에 의해 수행된다. JBD2 데몬의 커밋 인터벌은 include/linux/JBD2.h 파일에 5초로 정의되어 있으며, Ext4 파일시스템 마운트 시 옵션을 통해 변경 가능하다.

JBD2 데몬은 jbd2_journal_commit_트랜잭션() 함수를 통해 커밋과 체크포인트를 수행하며, 해당 함수는 대략적으로 다음과 같이 동작한다. 우선, Running 트랜잭션을 Committing 트랜잭션으로 변경하고, 저널 영역에서 체크포인트가 완료된 트랜잭션을 제거한다. 체크포인트 중인 트랜잭션들은 저널 슈퍼 블록 디스크립터에 있는 j_checkpoint_transaction 필드를 통해 양방향 리스트로 관리되며, 트랜잭션 내에 포함된 모든 메타데이터들이 파일 시스템 메타데이터 영역에 기록되면 해당 트랜잭션을 슈퍼 블록 디스크립터의 체크포인트 리스트에서 제거함으로써 해당 트랜잭션의 체크포인트가 완료한다.

그 다음 디스크립터 블록을 메모리에 할당하고, Committing 트랜잭션에 포함된 메타데이터의 위치 정보를 디스크립터 블록에 추가한다. 모든 메타데이터의 위치 정보가 디스크립터 블록에 추가되면, 디스크립터 블록과 메타데이터 블록을 순서대로 WRITE_SYNC 플래그가 설정된 bio를 구성하고 기존 요청에서 해당 bio의 Merge 여부에 따라 Plugged 리스트에 삽입하거나 입출력 스케줄러에 전달한다. 이후 커밋 블록에 대한 요청을 구성하여 입출력 스케줄러에 전달한다. 커밋 블록은 요청에 WRITE_FLUSH_FUA 플래그가 설정되어 입출력 스케줄러에 전달된다. 요청에 FLUSH 플래그가 설정되어 있으면 요청에 포함된 데이터를 스토리지에 기

록하기 전에 스토리지 내에 있는 write-back 캐시에 저장된 요청들을 디스크 표면에 기록한다. 이때 write-back 캐시에 있는 이전에 발행된 디스크립터 블록과 메타데이터 블록이 디스크 표면에 기록된다. 따라서 커밋 블록은 디스크립터 블록 및 메타데이터 블록 이후에 디스크에 기록되도록 순서가 보장되게 된다. 또한, 커밋 블록은 FUA 플래그가 설정되어 있으므로 디스크의 write-back 캐시를 거치지 않고 바로 디스크 표면에 기록되며, 이로써 Committing 트랜잭션의 커밋이 완료된다.

커밋이 완료되면 JB2 데몬은 Committing 트랜잭션을 저널 디스크립터의 체크포인트 트랜잭션 리스트에 추가함으로써 이후 JBD2 동작에서 체크포인트가 완료되도록 한다.

2.4.5 Recovery

시스템 장애가 일어나게 되면 저널 영역에 커밋된 변경사항이 담긴 기록을 읽어 실제 스토리지의 위치에 기록하는 간단한 복구 과정(Recovery)을 통해 일관성을 유지할 수 있다. 커밋하기 전에 시스템 장애가 발생한 트랜잭션의 경우에는 복구 시 무시하고, 시스템 장애 발생 전 커밋이 완료된 트랜잭션은 파일 시스템에 반영한다. 첫 번째 경우 파일 시스템에 대한 변경은 사라지지만 파일 시스템 상태의 일관성은 유지된다. 두 번째 경우 변경을 적용하므로 파일 시스템에 대한 입출력 데이터 전송이 제대로 종료되지 않아서 일관성이 깨진 상태를 해결한다.

### 2.5 Ext4 write() 시스템 콜

write()가 호출되면 Ext4 파일 시스템의 경우 Virtual File System(VFS)를 통해 ext4_file_write() 함수가 호출된다. ext4_file_write() 함수는 내부적으로 파일 개방 시 설정한 O_DIRECT 플래그 설정 여부에 따라 다른 방식으로 처리된다. O_DIRECT가 설정된 파일은 사용자 데이터를 페이지 캐시에 기록하고, 기록된 더티 페이지에 대해 블록 입출력 요청을 구성하여 더티 페이지가 디스크에 기록될 때까지 기다린다. O_DIRECT가 설정되지 않은 파일에 대해서는 버퍼링된 쓰기(Buffered write)가 수행된다. 버퍼링된 쓰기는 기록하고자 하는 사용자 데이터를 즉시 디스크에 기록하지 않으며, 페이지 캐시에 데이터를 기록하고 해당 페이지의 상태를 더티로 설정한다.

본 논문에서 allocating write()는 쓰기 요청에 대해 추가 블록을 할당 받는 write()이고, non-allocating write()는 추가 블록을 할당 받지 않는 write()라고 정의한다. allocating write() 시에는 블럭 비트맵과 아이노드(파일 크기)가 갱신된다. non-allocating write() 시에는 아이노드가 변경된다. non-allocating write()이지만 파일크기가 변경될 수 있다. 커널 버전별로 allocating write와 non-allocating write 시 수반되는 저널링 메커니즘에 큰 차이가 있다. 이를 자세히 살펴보기로 한다.

| Write Operation | Running Transaction's inode list | |
|---|---|---|
| | Allocating write() | Non-allocating write() |
| **write(f_1.txt)** | {} | {**$I_1$**} |
| **write(f_2.txt)** | {} | {$I_1$, **$I_2$**} |
| **write(f_3.txt)** | {} | {$I_1$, $I_2$, **$I_3$**} |
| **write(f_4.txt)** | {} | {$I_1$, $I_2$, $I_3$, **$I_4$**} |
| **write(f_5.txt)** | {} | {$I_1$, $I_2$, $I_3$, $I_4$, **$I_5$**} |
| **write(f_6.txt)** | {} | {$I_1$, $I_2$, $I_3$, $I_4$, $I_5$, **$I_6$**} |
| **write(f_7.txt)** | {} | {$I_1$, $I_2$, $I_3$, $I_4$, $I_5$, $I_6$, **$I_7$**} |
| **write(f_8.txt)** | {} | {$I_1$, $I_2$, $I_3$, $I_4$, $I_5$, $I_6$, $I_7$, **$I_8$**} |
| **write(f_9.txt)** | {} | {$I_1$, $I_2$, $I_3$, $I_4$, $I_5$, $I_6$, $I_7$, $I_8$, **$I_9$**} |

그림 3 트랜잭션의 아이노드 리스트(3.4.0 버전)
Fig. 3 Transaction's inode list(3.4.0 version)

커널 3.4.0 버전부터 살펴보기로 한다. non-allocating write() 시, 해당 파일의 아이노드를 running 트랜잭션의 아이노드 리스트에 추가한다. allocating write()의 경우는 완전히 다른 메커니즘이 사용된다. Ext4 파일 시스템은 페이지 캐시에 있는 내용을 디스크에 내리는 시점에 디스크 블록을 할당한다. allocating write의 경우 dirty된 페이지를 디스크에 내릴 때, 블록할당이 일어나고, 그 시점에 블록 비트맵과 갱신된 아이노드 블록이 running 트랜잭션에 추가된다(그림 3). non-allocating write의 경우는 write를 요청한 응용프로그램의 컨텍스트에서 갱신된 아이노드가 running 트랜잭션에 추가되는 반면 allocating write는 쓰기 요청과 아이노드의 저널 추가가 서로 다른 컨텍스트에서 이루진다.

그림 3은 리눅스 커널 3.4.0에서 allocating write와 non-allocating write간의 차이점을 도식적으로 표현하고 있다.

3.7.10 이후 버전에서는 아이노드를 트랜잭션의 아이노드 리스트에 추가함에 있어 allocating write와 non-allocating write가 동일한 메커니즘을 사용한다. “Ext4: remove calls to ext4_jbd2_file_inode() from delalloc write path by T Ts′o”[16] 패치가 적용되어 write() 시스템 콜 내에서는 변경된 아이노드를 Running 트랜잭션의 아이노드 리스트에서 삽입하지 않는다. 그림 4와 같이 non-allocating write() 시 Running 트랜잭션의 아이노드 리스트에 변경된 파일의 아이노드를 삽입하지 않는다.

## 3. Ext4 fsync() 분석

### 3.1 커널 3.4.0

fsync() 시스템 콜이 호출되면 fsync()의 인자로 주어진 파일의 더티 페이지를 Request로 구성하여 입출력

| Write Operation | Running Transaction's inode list | |
|---|---|---|
| | Allocating write() | Non-allocating write() |
| **write(f_1.txt)** | {} | {} |
| **write(f_2.txt)** | {} | {} |
| **write(f_3.txt)** | {} | {} |
| **write(f_4.txt)** | {} | {} |
| **write(f_5.txt)** | {} | {} |
| **write(f_6.txt)** | {} | {} |
| **write(f_7.txt)** | {} | {} |
| **write(f_8.txt)** | {} | {} |
| **write(f_9.txt)** | {} | {} |

그림 4 트랜잭션의 아이노드 리스트(3.8.0, 4,6,2 버전)
Fig. 4 Transaction's inode list(3.8.0, 4.6.2 version)

스케줄러에 전달하고, 해당 파일의 아이노드를 현재 Running 트랜잭션의 아이노드 리스트에 삽입한다. 만약 파일의 아이노드가 이미 아이노드 리스트에 있다면 중복해서 삽입하지 않는다.

그 다음, 파일의 메타데이터를 디스크 저널 영역에 기록하기 위해 JBD2 데몬을 깨운다. JBD2 데몬이 깨어나면 우선 fsync()의 인자로 주어진 파일의 메타데이터가 포함된 Running 트랜잭션을 Committing 트랜잭션으로 변경한다. 이 후 Committing 트랜잭션의 아이노드 리스트를 순회하여 각 아이노드에 해당하는 파일의 더티 페이지들을 Request로 구성하여 입출력 스케줄러에 전달한다.

allocating write() 동작 후 fsync()가 수행된 경우에는 트랙잰션의 아이노드 리스트에 아이노드가 포함되어 있지 않기 때문에, 그림 5와 같이 메타데이터 블록을 저널에 기록하기 전에 데이터를 디스크에 기록하지 않는다. 하지만 non-allocating write() 후 fsync()가 수행된다면, write() 동작 중 아이노드를 아이노드 리스트에 삽입하기 때문에 그림 6과 같이 JBD2 데몬에 의해 데이터가 기록된다. 해당 파일의 더티 페이지들이 현재 디스크에 기록 중인 상태거나 이미 기록이 완료된 상태라면 추가로 더티 페이지들에 대한 요청을 발생시키지 않는다. 여기서 디스크에 기록 중인 상태는 페이지 캐시에 있는 더티 페이지가 디스크 데이터 영역에 기록 중인 상태를 의미한다.

모든 파일의 더티 페이지들이 디스크에 기록되면, Committing 트랜잭션에 포함된 메타데이터의 상태를 JBDDirty로 변경하고 디스크립터 블록과 메타데이터 블록을 구성하여 디스크 저널 영역에 기록한다. 디스크 저널 영역에 디스크립터 블록과 메타데이터 블록을 기록한 후, 플러시 명령어를 전송하여 해당 블록들이 영속

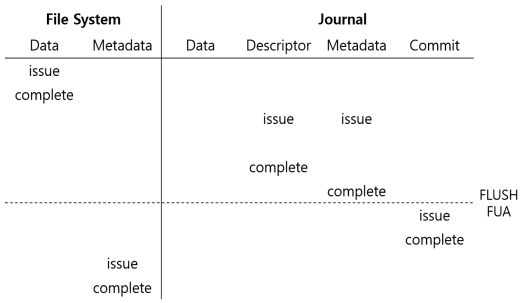


그림 5 블록 할당 시 fsync() 동작 순서(3.4.0 버전)
Fig. 5 fsync() flow after allocating write() (3.4.0 version)

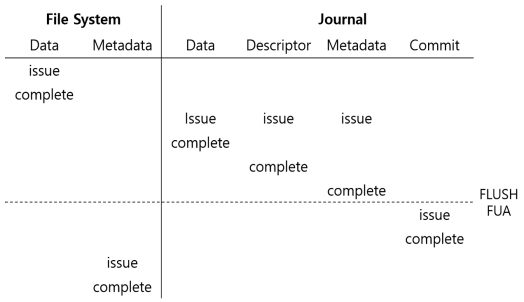


그림 6 블록 비 할당 시 fsync() 동작 순서(3.4.0 버전)
Fig. 6 fsync() flow after non-allocating write() (3.4.0 version)

적으로 디스크 표면이 저장되는 것을 확인한다. 추후 커밋 블록을 디스크의 저널 영역에 기록하여 해당 트랜잭션의 커밋을 완료하고 Committing 트랜잭션 내에 포함된 메타데이터의 상태를 JBDDirty에서 Dirty로 변경한다. 커밋이 완료되어 Dirty 상태로 변경된 메타데이터들은 이후 Flusher 쓰레드인 Pdflush에 의해 디스크의 메타데이터 영역에 체크포인트 된다.

### 3.2 커널 3.8.0

커널 3.8.0 버전부터는 다른 알고리즘이 사용된다. 3.8.0~4.6.1 버전에서는 ordered 모드 저널링을 사용함에도 불구하고, 데이터와 저널사이에 쓰기 순서가 역전될 수 있는 버그가 존재하였다.

3.8.0 버전부터 적용된 "Ext4: remove calls to ext4_jbd2_file_inode() from delalloc write path by T Ts′o" [16] 패치는 write() 시스템 콜과 Kworker 쓰레드 동작 중 변경된 파일의 아이노드를 Running 트랜잭션의 아이노드 리스트에 추가하는 코드를 제거하였다. 따라서 3.8.0 버전은 write()에 의해 변경된 파일의 아이노드가 트랜잭션의 아이노드 리스트에 삽입되지 않기 때문에, 그림 7과 같이 더티 페이지들이 JBD2 데몬에 의해 디스크에 기록되지 않는다. fsync() 동작 중 데이터가

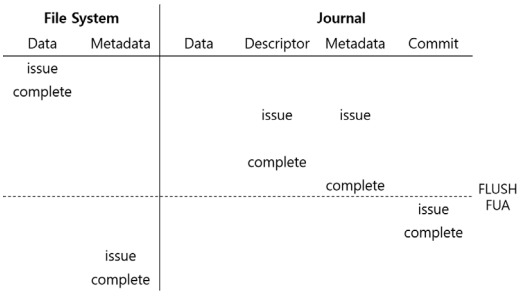


그림 7 fsync() 동작 순서(3.8.0, 4.6.2)
Fig. 7 fsync() flow(3.8.0, 4.6.2 version)

디스크에 기록되지 않기 때문에, 3.8.0 이전 버전과 비교해 non-allocating write() 동작 후 수행되는 fsync()의 지연시간이 단축되었다.

하지만, 3.8.0 버전은 “Ext4: Fix data exposure after a crash by Jan Kara”[17]에서 데이터와 메타데이터가 기록되는 순서가 바뀌어 Ordered 모드가 깨지는 현상이 보고되었다. 해당 버그는 “Ext4: remove calls to ext4_jbd2_file_inode() from delalloc write path by T Ts′o” [16] 패치로 인해 트랜잭션의 아이노드 리스트에 아이노드가 삽입되지 않기 때문이다. 변경된 파일의 더티 페이지가 디스크에 기록되기 전에 JBD2 데몬이 메타데이터를 기록하여 데이터와 메타데이터의 기록 순서가 바뀌는 경우 발생한다.

### 3.3 커널 4.6.2

Kernel 4.6.2 버전부터 3.8.0 버전의 ordered 모드 결함을 해결한 “Ext4: Fix data exposure after a crash by Jan Kara”[17] 패치가 적용되었다. 해당 패치는 변경된 파일의 아이노드가 Kworker 쓰레드에 의해 Running 트랜잭션의 아이노드 리스트에 삽입되도록 변경함으로써 JBD2 데몬이 메타데이터를 저널에 기록하기 전에 데이터가 기록되는 것을 보장한다. fsync()의 동작 순서는 3.8.0 버전의 fsync() 동작 순서인 그림 7과 같다.

## 4. 분석

### 4.1 테스트 코드

그림 8은 테스트 프로그램의 의사코드이다. file0~file9 10개의 파일을 생성 및 개방한 후 각 파일에 대해 3KB의 write operations 수행한다. 마지막에 file9 파일에 대해 fsync()를 호출한다. 커널 소스 코드 내 블록 입출력 디버그 코드, Blktrace, 그리고 저널 파일의 덤프를 통해 테스트 프로그램 동작을 분석하였다. 본 실험은 fsync() 동작에 차이를 보이는 리눅스 3.4.0, 3,8,0, 4.6.2 세 개 버전에서 수행하였다. 쓰기연산은 allocating write와 non-allocating write 두 가지에 대해 수행하였다.

```
procedure testFsync()
  for file = file0; file <= file9; file++ do
    f[idx] = open(file, O_RDWR, O_CREATE);
    lseek(f[idx], 0, SEEK_END);
    write(f[idx], buf, 3 * 1024);
    if file == file9
      fsync(f[idx]);
    close(f[idx]);
  end for
end procedure
```

그림 8 테스트 프로그램 의사코드
Fig. 8 Pseudo-code of the test program

### 4.2 커널 3.4.0

4.2.1 allocating write()

Blktrace의 출력 결과는 첫 번째 열부터 시간, Action, RWBS, 섹터 + 크기를 각각 나타낸다. Action의 D는 Dispatch의 약자이다. 디스패치란 입출력 스케줄러가 관리하는 디스패치 큐에 있는 요청이 디바이스 드라이버에 의해 선택되었음을 의미한다. RWBS는 최소한 하나의 문자를 포함하는 문자열로, 해당 요청의 ‘R’ 읽기, ‘W’ 쓰기, ‘B’ 베리어, ‘S’ Synchronous의 동작을 나타낸다. 예를 들어, 아래의 출력 결과에서 ②의 ‘WS’는 해당 요청이 Synchronous 쓰기 요청임을 의미한다.

아래의 출력 결과를 자세히 보도록 하자. ① 라인은 fsync()의 인자로 받은 file9 데이터를 파티션 내 그룹 1 데이터 영역에 기록하는 동작이다. LBA 187310104는 파티션 내 두 번째 그룹(Group 1)의 데이터 영역에 속해 있으며 테스트 프로그램에 의해 fsync() 함수의 인자로 받은 f_9.txt 파일의 데이터가 디스크의 그룹 1 데이터 영역에 in-place 업데이트 된 것을 의미한다. ②의 LBA 216670728는 파티션 내 저널 영역에 속해 있는 블록이며, 20KB(40 * 512(섹터)), 즉 5개의 블록이 저널 영역에 기록되었다. 첫 번째(4KB) 블록은 디스크립터 블록이며, 나머지 네 개(16KB)의 블록은 f_0.txt, …,f_9.txt의 아이노드가 포함된 그룹 0의 그룹 디스크립터 테이블, f_9.txt의 데이터 블록 할당에 따른 그룹 1의 블록 비트맵, f_0.txt,…,f_9.txt 10개 파일에 대한 두 개의 그룹 0 아이노드 테이블이다. ②의 LBA 216670768는 커밋 블록이며, 디스크립터 블록과 메타데이터 블록 뒤에 기록되어 해당 트랜잭션의 커밋 후 fsync()의 동작이 완료되었음을 의미한다. ③에서는 Flusher 쓰레드인 Pdflush에 의해 f_0.txt,…,f_8.txt 9개 파일의 데이터가 Group 1의 데이터 영역에 기록되었음을 의미한다. ④의 LBA 216670776는 파티션의 저널 영역에 해당하며, 총 5개의 블록이 기록되었다. 첫 번째 블록은 저널 디스크립터 블록이고, 나머지 4개 블록은 각각 f_0.tx

t,…,f_8.txt의 아이노드가 포함된 그룹 0의 그룹 디스크립터 테이블, f_0.txt,…,f_8.txt의 데이터 블록 할당에 따른 Group 1의 블록 비트맵, f_0.txt,…,f_8.txt 9개 파일에 대한 두 개의 그룹 0 아이노드 테이블이다. ④의 LBA 216670816는 커밋 블록이며, 해당 트랜잭션의 커밋이 완료되었음을 알 수 있다. 마지막으로 ⑤은 ④에서 저널 영역에 기록한 그룹 0의 그룹 디스크립터 테이블, Group 1의 블록 비트맵, 두 개의 그룹 0 아이노드 테이블이 Pdflush에 의해 그룹 1의 메타데이터 영역에 기록되어 체크포인트 되었음을 의미한다.

| | |
|---|---|
| 0.000011188 D WS 187310104 + 8 [fsync] | ① |
| 0.000195809 D WS 216670728 + 40 [jbd2/sda5-8]<br>0.001254168 D WS 216670768 + 8 [swapper/0] | ② |
| 30.003263883 D W 187306096 + 8 [flush-8:0]<br>30.003337075 D W 187306104 + 8 [flush-8:0]<br>30.003419284 D W 187320256 + 8 [flush-8:0]<br>30.003513664 D W 187320264 + 16 [flush-8:0]<br>30.003608259 D W 187320280 + 24 [flush-8:0]<br>30.003752944 D W 187320312 + 8 [flush-8:0] | ③ |
| 35.195248762 D WS 216670776 + 40 [jbd2/sda5-8]<br>35.196295633 D WS 216670816 + 8 [swapper/0] | ④ |
| 65.203244067 D W 187044104 + 16 [flush-8:0]<br>65.203303967 D W 187035656 + 8 [flush-8:0]<br>65.203443859 D W 187043856 + 8 [flush-8:0] | ⑤ |

4.2.2 non-allocating write()

①의 LBA 187310104는 파티션 내 그룹 1의 데이터 블록 영역에 속하며, fsync()의 인자로 전달 받은 f_9.txt의 데이터가 그룹 1의 데이터 블록에 기록되었음을 의미한다. ②의 LBA 187320312~187320296까지는 그룹 1의 데이터 블록 영역에 해당하며, JBD2 데몬에 의해 write ()가 수행된 f_0.txt,...,f_8.txt의 9개 파일의 데이터가 그룹 1의 데이터 블록 영역에 기록되었다. ③의 LBA 216670824는 파티션 내 저널 영역이며, 세 개의 블록이 저널 영역에 기록되었음을 알 수 있다. 첫 번째 블록은 저널 디스크립터 블록이며, 나머지 두 개 블록은 f_0.txt,...,f_9.txt 10개 파일의 아이노드가 포함된 두 개의 아이노드 테이블이다. 216670848은 커밋 블록이며, 메타데이터 블록 뒤에 기록되어 해당 트랜잭션의 커밋 후 fsync()의 동작이 완료되었음을 의미한다. 마지막으로 ④의 LBA 187044104는 그룹 0의 아이노드 테이블 영역에 속하며, ③에서 저널 영역에 기록된 f_0.txt,...,f_9.txt 10개 파일에 대한 두 개의 아이노드 테이블이 Kworker 쓰레드에 의해 그룹 0의 아이노드 테이블에 기록되어 체크포인트 되었음을 의미한다.

| | |
|---|---|
| 19.867466200 D WS 187310104 + 8 [fsync] | ① |
| 19.867663429 D WS 187320312 + 8 [jbd2/sda5-8]<br>19.867723537 D WS 187320296 + 8 [jbd2/sda5-8]<br>19.867855738 D WS 187320256 + 40 [jbd2/sda5-8]<br>19.868121285 D WS 187306096 + 16 [jbd2/sda5-8] | ② |
| 19.867959390 D WS 216670824 + 24 [jbd2/sda5-8]<br>19.869192579 D WS 216670848 + 8 [swapper/0] | ③ |
| 54.859238145 D W 187044104 + 16 [flush-8:0] | ④ |

### 4.3 커널 3.8.0과 커널 4.6.2

Allocating write()의 경우에는 Kernel 3.4.0 버전의 트레이스 결과와 유사하며 non-allocating write()의 경우에 아래와 같이 다른 트레이스 결과를 보인다. ①은 커널 3.4.0 버전의 non-allocating write() 동작과 같으며, 3.8.0 버전에서는 3.4.0 버전의 ②와 같이 JBD2 데몬에 의해 f_0.txt,...,f_8.txt의 9개 파일의 데이터가 저널 영역에 기록되지 않는다. ②는 3.4.0 버전의 ③과 같이 트랜잭션을 저널 영역에 기록한 후, 해당 트랜잭션을 커밋하여 fsync()의 동작을 완료한다.

3.4.0 버전의 non-allocating write() 시 JBD2 데몬에 의해 기록된 f_0.txt,...,f_8.txt의 9개 파일의 데이터가 3.8.0 버전의 non-allocating write()인 경우에는 ③과 같이 Kworker 쓰레드에 의해 asynchronous I/O로 데이터 영역에 기록된다. 즉, f_0.txt,...,f_8.txt의 9개 파일의 데이터가 JBD2 데몬에 의해 fsync() 동작 중에 데이터 영역에 기록되지 않기 때문에, fsync() 지연시간이 단축되었다.

Kernel 4.6.2 버전은 3.8.0 버전의 트레이스 결과와 동일하다.

| | |
|---|---|
| 164.246733939 D WS 187326472 + 8 [fsync] | ① |
| 164.246907477 D WS 216658232 + 24 [jbd2/sda5-8]<br>164.247915597 D WS 216658256 + 8 [swapper/0] | ② |
| 194.249682731 D W 187306088 + 8 [flush-8:0]<br>194.249753298 D W 187306096 + 8 [flush-8:0]<br>194.249848189 D W 187306104 + 40 [flush-8:0]<br>194.249953993 D W 187306144 + 16 [swapper/0]<br>194.250145763 D W 187044104 + 16 [swapper/0] | ③ |
| 199.553765728 D WS 216658264 + 8 [swapper/0] | ④ |

### 4.4 전체 요약

그림 9는 커널 버전 별 10개 파일에 대한 테스트 프로그램의 실행 결과를 나타낸다. 3.4.0 버전의 리눅스 커널에서는 allocating write() 시 fsync()에서 D9와 M9가 디스크의 데이터 영역 및 저널 영역에 기록된다. 하지만, non-allocating write() 시 fsync()에서는 D9과 M9뿐만 아니라 write()만 수행된 D0,…,D8 및 M0,…,

| Kernel | Block Allocating | Result of fsync() | |
|---|---|---|---|
| | | Process Context | JBD2 Context |
| 3.4.0 | O | {$D_9$} | {$M_9$} |
| | X | {$D_9$} | {$D_0$ ~ $D_8$}, {$M_0$ ~ $M_9$} |
| 3.8.0 | O | {$D_9$} | {$M_9$} |
| | X | {$D_9$} | {$M_9$} |
| 4.6.2 | O | {$D_9$} | {$M_9$} |
| | X | {$D_9$} | {$M_9$} |

D# : Data of # / M# : Metadata of #

그림 9 커널 버전 별 fsync() 실행 결과
Fig. 9 fsync() result according to the kernel version

M8이 디스크의 데이터 영역 및 저널 영역에 기록된다.

커널 3.8.0 버전에서 allocating write() 시 fsync()의 동작은 3.4.0 버전의 allocating write() 시 fsync() 실행 결과와 동일하다. 하지만 non-allocating write() 시 fsync()의 동작은 D9와 M9만 디스크에 기록된다. 커널 4.6.2와 4.7.4 버전은 3.8.0 버전에서의 fsync() 실행 결과와 동일하다.

## 5. 성능 평가

### 5.1 실험 환경

실험은 PC와 Nexus 5에서 수행되었다. PC는 리눅스 커널 3.4.0, 3.8.0, 4.6.2 세 개 버전에서 그리고 Nexus 5는 3.4.0 버전에서 수행되었다.

| | | |
|---|---|---|
| PC | Hardware | CPU: Intel(R) i7-4770<br>RAM: 6 GByte<br>Storage: SSD 128GB |
| | OS | Ubuntu 14.04 LTS<br>Ubuntu 16.04 LTS |
| | Kernel | Linux kernel 3.4.0<br>Linux kernel 3.8.0<br>Linux kernel 4.6.2 |
| Mobile (Nexus 5) | Hardware | CPU: Qualcomm Snapdragon™ 800<br>RAM: 2 GByte<br>Storage: eMMC 16GB |
| | OS | Android 4.4 KitKat |
| | Kernel | Linux kernel 3.4.0 |

### 5.2 테스트 프로그램

본 실험은 fsync()의 동작을 확인하기 위해 1, 10, 100, 1000개의 파일을 개방한 후 각 파일에 대해 2KB의 쓰기 동작을 수행하고, 마지막으로 개방한 하나의 파일에 대해서만 fsync()를 호출하는 테스트 프로그램을 사용하였다. 각 실험은 allocating write()와 non-allocating write()인 경우를 구분하여 실험하였다. 실험 결과는 fsync() 작업의 지연 시간과 fsync() 작업에 의해 디스크 데이터 영역에 기록되는 데이터 블록의 개수를 성능 평가의 기준으로 삼아 실험을 진행하였다.

### 5.3 실험 결과

그림 10의 두 개 그래프는 fsync()의 지연시간을 나타낸 그래프이다. allocating write()의 경우 그림 10의 오른쪽 그래프와 같이 커널 버전과 무관하게 파일 개수가 증가함에 따라 fsync() 지연시간도 비례해서 증가하는 것을 확인할 수 있다. write()가 수행된 파일의 개수가 증가하면 JBD2 데몬이 저널에 기록하는 메타데이터의 개수도 증가하기 때문에, fsync()의 지연시간은 증가하게 된다. 파일의 개수가 1~100개일 때 보다 1000개일 때 fsync() 지연시간의 증가폭이 큰 이유는 1000개일 때 JBD2 데몬에 의해서 저널에 기록되는 메타데이터 블록 수가 100개일 때 대비 7배 이상 증가했기 때문이다.

non-allocating write()의 경우 그림 10의 왼쪽 그래프와 같이 3.4.0 버전은 파일 개수를 1개부터 1000개까지 증가시켰을 때 fsync()의 지연시간이 최소 30%에서 최대 1100% 가량 증가하는 것을 볼 수 있다. 3.4.0 버전에서는 non-allocating write()의 경우 변경된 파일의 아이노드를 트랜잭션의 아이노드 리스트에 삽입하여, fsync() 동작 중 JBD2 데몬에 의해 write()가 수행된 파일의 더티 페이지를 디스크에 기록하기 때문이다.

그림 11은 fsync()에 의해 기록된 데이터 블록의 개수를 나타낸 그래프이다. allocating write()의 경우 그림 11의 왼쪽 그래프와 같이 모든 커널 버전에서 동일하게 하나의 데이터 블록이 디스크에 기록된다. 이는 allocating write()되는 파일의 개수와 관계없이 항상 fsync()의 인자로 전달된 파일의 데이터 블록만 기록되기 때문이다.

Non-allocating write()인 경우 그림 11의 오른쪽 그래프와 같이 3.8.0 버전과 4.6.2 버전은 allocating write()와 같은 결과를 나타내며, 이를 통해 fsync()의 인자로 전달된 파일의 더티 페이지만 디스크에 기록되는 것을 알 수 있다. 하지만, 3.4.0 버전의 경우 allocating write()와 달리 파일의 개수가 증가할수록 fsync()에서 디스크에 기록되는 데이터 블록의 개수가 증가하는 것을 확인할 수 있다. 이는 non-allocating write()된 파일들의 아이노드가 트랜잭션의 아이노드 리스트에 삽입되기 때문이다. 즉, fsync() 함수에서 트랜잭션을 커밋할 때 아이노드 리스트에 존재하는 모든 아이노드의 더티 페이지를 디스크에 기록하기 때문이다.

## 6. 결 론

open(), write(), 그리고 fsync()를 사용하는 테스트 프로그램을 통해 커널 3.4.0, 3.8.0, 4.6.2 세 개 버전에서 fsync()의 동작을 분석하였다. 그 결과 세 개 버전 모두에서 write() 시 파일에 새로운 블록의 할당 유무에 따

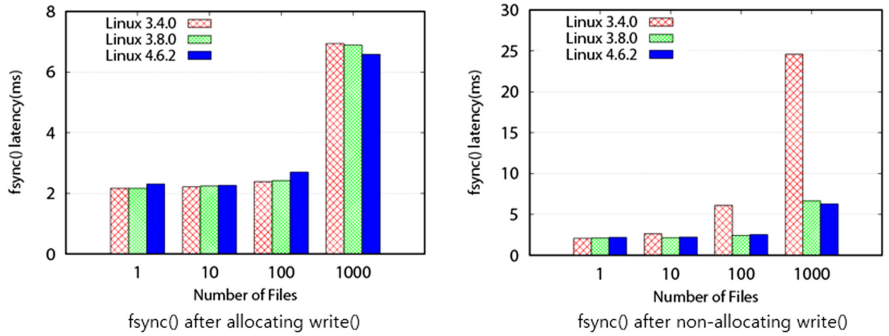


그림 10 fsync() 지연시간
Fig. 10 fsync() latency

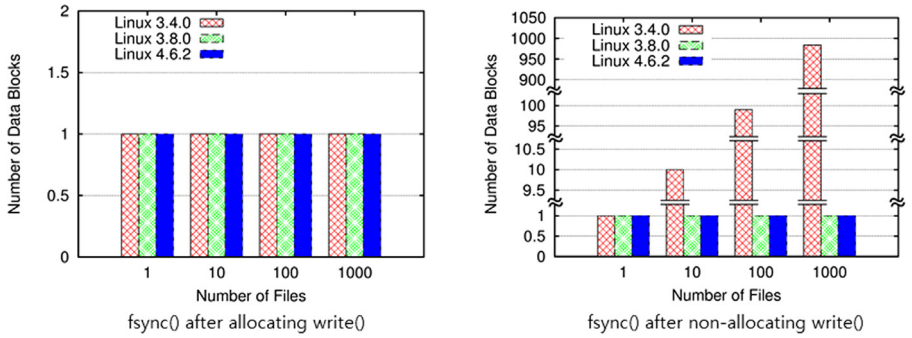


그림 11 fsync()에 의해 기록된 데이터 블록 수
Fig. 11 Number of written data blocks

라 테스트 프로그램의 함수 호출 경로 및 Blktrace 출력 결과가 다른 것을 확인하였다. 커널 3.4.0 버전은 non-allocating write()의 경우 write() 동작에 의해 변경된 파일의 아이노드가 Running 트랜잭션의 아이노드 리스트에 추가되어 파일 f_0.txt~f_8.txt의 더티 페이지가 JBD2 쓰레드에 의해 디스크의 데이터 영역에 기록된다. 하지만, 3.8.0 버전에서 변경된 파일의 아이노드가 아이노드 리스트에 삽입되지 않는다. 따라서 JBD2 데몬이 메타데이터를 디스크의 저널 영역에 기록하기 전에 Kworker 쓰레드에 의해 더티 페이지가 디스크에 기록될 때까지 기다릴 수 없게 되어 순차 모드를 보장할 수 없는 문제를 가지게 되었다.

커널 4.6.2 버전은 3.8.0 버전과 달리 Kworker 쓰레드에 의해 변경된 파일의 아이노드가 트랜잭션의 아이노드 리스트에 삽입되도록 변경 되었다. 따라서 3.8.0~4.6.1 버전이 가지고 있던 순차 모드의 결함이 해결 되었다. 또한, 커널 3.4.0과 4.6.2 버전에서는 non-allocating write() + fsync() 시에 변경된 파일의 메타데이터를 디스크의 저널 및 데이터 영역에 기록하지 않아 3.4.0 버전에 비해서 fsync()의 지연시간이 단축되었다.

SQLite와 같이 별도의 저널링을 지원하는 데이터베이스는 주기적으로 fsync()를 호출하여 저널 파일의 즉시 내구성을 보장하기 때문에[4], fsync()의 지연 시간이 비교적 긴 3.8.0 이전 버전보다 4.6.1 이후 버전을 사용하는 것이 데이터베이스와 유저 애플리케이션의 응답성면에서 더 효율적이다. 하지만, 커널 3.8.0~4.6.1 버전에서는 순차 모드의 결함으로 인해 데이터베이스에서 갱신된 모든 저널 파일에 대해 fsync()를 호출하지 않으면, 데이터베이스의 일관성이 깨지는 문제가 발생할 수 있다. 데이터베이스에서 하나의 저널 파일을 사용한다면, 저널 파일에 대해 주기적으로 fsync()가 수행되기 때문에 3.8.0~4.6.1 버전에서의 일관성이 깨지는 문제가 발생하지 않아 다른 커널 버전에 비해 더 효과적일 수 있다.

## References

[1] H. Kim, J. Kim, "Tuning the Ext4 Filesystem Performance for Android-based Smartphones," *Proc. of Frontiers in Computer Education. Springer Berlin Heidelberg*, pp. 745-752, 2012.

[2] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, "The New Ext4 Filesystem: Current Status and

Future Plans," *Proc. of the Linux symposium*, Vol. 2, 2007.
[3] S. Jeong, K. Lee, S. Lee, S. Son, Y. Won, "I/O stack Optimization for Smartphones," *Proc. of the USENIX Annual Technical Conference (USENIX ATC 13)*, 2013.
[4] W.H. Kim, B. Nam, D. Park, Y. Won, "Resolving Journaling of Journal Anomaly in Android I/O: Multi-version B-tree with Lazy Split," *Proc. of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014.
[5] D. Jeong, Y. Lee, J. Kim, "Boosting Quasi-asynchronous I/O for Better Responsiveness in Mobile Devices," *Proc. of 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.
[6] D. Kang, Y. Eom, "FS-LRU: A Page Cache Algorithm for Eliminating Fsync Write on Mobile Devices," *Proc. of IEEE International Conference on Consumer Electronics (ICCE), IEEE*, 2016.
[7] Y. Kang, D. Shin, Y Kang, D Shin, "Per-block-group Journaling for improving Fsync Response Time," *Proc. of The 18th IEEE International Symposium on Consumer Electronics (ISCE 2014), IEEE*, 2014.
[8] D. Park, M. Kim, D. Shin, "Optimizing Fsync Performance with Dynamic Queue Depth Adaptation," *Journal of Semiconductor Technology and science*, Vol. 15, No. 5, pp. 571-577, Oct. 2008.
[9] D. Park, D. Shin, "Fine-Grained Journaling for Reducing Fsync System Call Latency," *Proc. of the KIISE Winter Conference*, pp. 1219-1221, 2015. (in Korean)
[10] D. Bovet, M. Cesati, Understanding the Linux Kernel Edition, O'Reilly, Sebastopol, CA. 2006.
[11] M. Bjørling, J. Axboe, D. Nellans, P. Bonnet, "Linux Block IO: Introducing Multi-Queue SSD Access on Multi-core Systems," *Proc. of the 6th International Systems and Storage Conference, ACM*, 2013.
[12] J. Watkins, D. Labuda, W. Loo, "Input/Output Cache with Mapped Pages Allocated for Caching Direct (Virtual) Memory Access Input/Output Data based on Type of I/O Devices." U.S. Patent No. 5,263,142. 16 Nov. 1993.
[13] J. Coke, A. Bhatt, S. Graham, D. Lent, "Implementing scatter/gather operations in a direct memory access device on a personal computer," U.S. Patent No. 5,708,849, 1998.
[14] K. Fairbanks, C. Lee, Y. Xia, "Timekeeper: A Metadata Archiving Method for Honeypot Forensics," *Information Assurance and Security Workshop*, 2007, IAW'07, IEEE SMC, 2007.
[15] S. Tweedie, "Journaling the Linux Ext2fs Filesystem," The Fourth Annual Linux Expo. 1998.
[16] T. Ts'o, (2012, Nov. 16). Ext4: Remove Calls to ext4_jbd2_file_inode() from Delalloc Write Path [Online]. Available: http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=f3b59291a69d0b734be1fc8be489fef2dd846d3d
[17] J. Kara, (2016, Jan. 11). Ext4: Fix Data Exposure after a Crash [Online]. Available: https://patchwork.ozlabs.org/patch/565741

**손 성 배**
2016년 금오공과대학교 컴퓨터공학부 졸업(학사). 2016년 한양대학교 컴퓨터소프트웨어학과 석사 재학 중. 관심분야는 운영체제, 파일 시스템, 리눅스 블록 입출력

**노 연 진**
2016년 한양대학교 컴퓨터공학부 졸업(학사). 2016년 한양대학교 컴퓨터소프트웨어학과 석사 재학 중, 관심분야는 운영체제, 메모리 관리, 파일 시스템

**이 도 근**
2009년 한양대학교 전자통신컴퓨터공학부 졸업(학사). 2012년 한양대학교 컴퓨터소프트웨어학과 석박사 통합과정 수료 관심분야는 운영체제, 메모리 시스템, 비휘발성 메모리

**박 성 순**
1984년 홍익대학교 전자계산학과(학사) 1994년 고려대학교 전산학과(박사). 현재 안양대학교 컴퓨터공학과 교수. (주)글루시스 대표이사. 관심분야는 스토리지, 파일시스템, 클라우드컴퓨팅

**원 유 집**
1990년 서울대학교 계산통계학과(학사) 1992년 서울대학교 계산통계학과(석사) 1997년 University of Minnesota 전산학(박사). 현재 한양대학교 컴퓨터소프트웨어학과 교수. 관심분야는 운영체제, 파일 시스템, 스토리지 시스템, 모바일 스토리지